Java Virtual Machine Exposed

Session 647

These are confidential sessions—please refrain from filming or taking pictures.

Enterprise IT

Java Virtual Machine Exposed

Roger Hoover, Mikey McDougall, and Victor Hernandez
Java Engineering Team

http://developer.apple.com/wwdc2005/

This Talk Is for Developers Who Want

- To look under the hood at the Java VM engine on Tiger
- To kick the tires of the Java VM on Intel-based Macs

What You'll Learn About Mac OS X Java

- The Here (Tiger)
  - New Java support in Tiger
    - J2SE 5.0 with C1 compiler
- The Now (Intel-based Macs)
  - Java support for Intel-based Macs
    - Java 1.4.2, J2SE 1.5.0, C1 compiler
  - Introducing the C2 compiler for Intel-based Macs
- The Race Ahead
  - Snapshot of our current performance numbers

Java Virtual Machine in Tiger

Mikey McDougall
Java Product Engineering Team

http://developer.apple.com/wwdc2005/

Java on Tiger

- Tiger includes Java 1.4.2 and Java 1.3.1
- J2SE 5.0
  - Released simultaneously with Tiger
  - Available NOW as a software download
  - Planned for later this year as a Software Update

Mac OS X Support for Java

- J2SE 5.0 is the focus of Apple's Java efforts
- Java 1.4 applications just run in J2SE 5.0
  - Report backward-compatibility bugs
- Java 1.3.1 is going away

“Your application already works on 5.0.”

“The platform was subjected to a higher level of testing than any previous version, to prepare it for mission-critical production.”

Calvin Austin, Sun Microsystems, *Five Reasons to Move to the J2SE 5 Platform*

What's New in J2SE 5.0

Biggest changes in Java's history

Language changes

Autoboxing	Generics	Static Import
Enhanced for loop	varargs	Metadata

Library and runtime changes

Concurrent Programming	Memory Model
JVMTI	Monitoring/Management

J2SE 5.0 Language Features

- Autoboxing
  - automatic conversion between types
    - Unboxed (int, float, etc)
    - Boxed (Integer, Float, Etc)
- Generics
  - Support for parameterized classes
    - Similar to templates in C++
  - javac type-checks and generates casts
- Static Import
  - Imports static methods and fields of classes/interfaces

J2SE 5.0 Language Features

- Enhanced `for` loop
  - Easy iterators for arrays and Iterable types
- Variable argument methods (varargs)
- Metadata
  - tag Java declarations with additional information

It's All Done in javac

- Most of these features are syntactic sugar implemented in javac that
  - Makes your source code cleaner
  - Enables better code reuse
  - Mostly does not effect performance
- But be aware that autoboxing creates objects for you…

Over Autoboxing

Here is a program that calculates pi using the Gregory series

```
Double quarterPi = 1.0;
for (int i = 3 ; i < 1000000;) {
    quarterPi -= 1.0/i;
    i += 2;
    quarterPi += 1.0/i;
    i += 2;
}
System.out.println(" pi= " + (quarterPi*4));
```

Over Autoboxing

Here is a program that calculates pi using the Gregory series.

```
Double quarterPi = 1.0;
for (int i = 3 ; i < 1000000;) {
    quarterPi -= 1.0/i;
    i += 2;
    quarterPi += 1.0/i;
    i += 2;
}
System.out.println(" pi= " + (quarterPi*4));
```

Here is another version (2.7 times faster)

```
double quarterPi = 1.0;
for (int i = 3 ; i < 1000000;) {
    quarterPi -= 1.0/i;
    i += 2;
    quarterPi += 1.0/i;
    i += 2;
}
System.out.println(" pi= " + (quarterPi*4));
```

Over Autoboxing

Here is a program that calculates pi using the Gregory series

```
Double quarterPi = 1.0;
for (int i = 3 ; i < 1000000;) {
    quarterPi -= 1.0/i;
    i += 2;
    quarterPi += 1.0/i;
    i += 2;
}
System.out.println(" pi= " + (quarterPi*4));
```

Here is another version (2.7 times faster)

```
double quarterPi = 1.0;
for (int i = 3 ; i < 1000000;) {
    quarterPi -= 1.0/i;
    i += 2;
    quarterPi += 1.0/i;
    i += 2;
}
System.out.println(" pi= " + (quarterPi*4));
```

Can you spot the difference?

Over Autoboxing

The first example is using a `Double` and javac is autoboxing the value

```
Double quarterPi = Double.valueOf(1.0D);
for(int i = 3; i < 1000000;) {
    quarterPi = Double.valueOf(quarterPi.doubleValue() - 1.0D / (double)i);
    i += 2;
    quarterPi = Double.valueOf(quarterPi.doubleValue() + 1.0D / (double)i);
    i += 2
}
System.out.println(" pi= " + (quarterPi.doubleValue()*4));
```

The second uses a `double` and is unchanged

J2SE 5.0 Library and Runtime Features

- Concurrent Programming
  - Finer-grained synchronization for multi-threaded apps
  - Import java.util.concurrent.*
- Java Memory Model
  - Guarantees for reading and writing shared memory in multi-threaded applications
- JVMTI
  - Programming interface for development and monitoring tools
  - Replaces JVMPI and JVMDI

J2SE 5.0 Improvements

- High Precision Time
  - `System.nanoTime()` provides nanosecond granularity
- StringBuilder (vs StringBuffer)
  - Not thread-safe
  - Zero synchronization cost
  - String's + operator updated to use StringBuilder
- Graphics improvements discussed at yesterday's Java Performance talk

HotSpot Java Virtual Machine

- HotSpot JVM provides
  - C1 Just-in-Time compiler
  - Automatic G5 optimization
  - Class Data Sharing between JVM instances
  - Variety of Garbage Collection algorithms

HotSpot C1 Compiler

- Goal
  - Improve Java performance over Interpreter
  - Without dramatically increasing startup time or footprint
- How?
  - Dynamically compiles a program's "hot" methods
- Result: 10x faster than Interpreter
- Compilation takes time
  - Balances compiling and running
  - Classic compiler optimizations require lots of time and space
  - Optimal register allocation is expensive
- C1 compiler is default

C1 Compiler

- Runtime support for dynamic compilation
  - On-Stack Replacement
  - Inlining of virtual methods
  - Deoptimization
- Object-oriented optimizations
  - Invocation cache for virtual methods
  - Optimized class hierarchy operations (`instanceof` and `checkcast`)
- Code generation optimized for PowerPC
  - AltiVec instructions for memory operations
  - 64-bit G5 instructions for longs, doubles and floats

Faster Math on G5

- 1.4.2 and 1.5.0 use 64-bit registers and instructions for long math when running on G5

```
Java source:
static long longAdd(long a, long b)
{
  return a+b;
}
```

```
Compiled on G5:
0000: ld  r10,0x40(sp)
0004: ld  r9,0x38(sp)
0008: add r14,r10,r9
000c: blr
```

```
Compiled on G4:
0000: lwz   r9,0x40(sp)
0004: lwz   r10,0x44(sp)
0008: lwz   r7,0x38(sp)
000c: lwz   r8,0x3C(sp)
0010: addc  r15,r10,r8
0014: adde  r14,r9,r7
0018: blr
```

Class Data Sharing

- Caches internal representation of classes on disk
- Benefits
  - Reduces class load time in later invocations of VM
  - Improves startup time
- Can be controlled on the command line by
  - `-Xshare:auto` (default)
  - `-Xshare:off`
  - `-Xshare:on`
- Originally innovated by Apple since Java 1.3
  - Now part of Sun's J2SE 5.0
  - Tiger creates the shared archive for you by default

Garbage Collection in HotSpot

- Three Generational GC algorithms to choose from
- Meet specific needs

Algorithm	Benefit
Serial Collector (default)	Supports Class Data Sharing
Concurrent Mark and Sweep	Scales well for large heaps
Parallel Scavenge	Reduces GC pause time

Tip: Track your GC activity with -verbose:gc

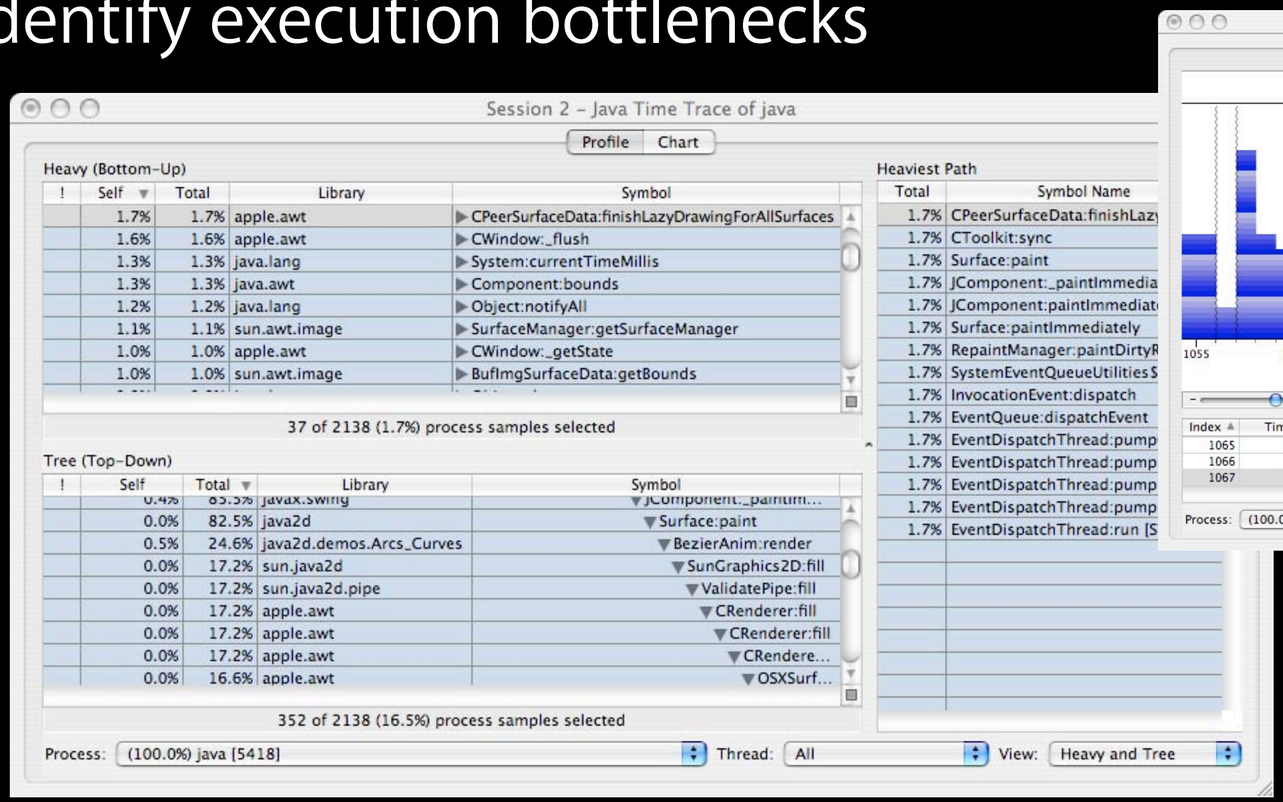
Shark to profile your Java application

- Use Shark to
  - Identify hot allocations
  - Identify hot methods
- Works with all Java versions (J2SE 5.0, Java 1.4.2, Java 1.3.1)
- Add -XrunShark to your Java flags
- Attach to your Java application

Optimizing Java

Shark for Java

- Identify execution bottlenecks



Swimming with Shark

Session 407
Friday 2:00PM

Hands On

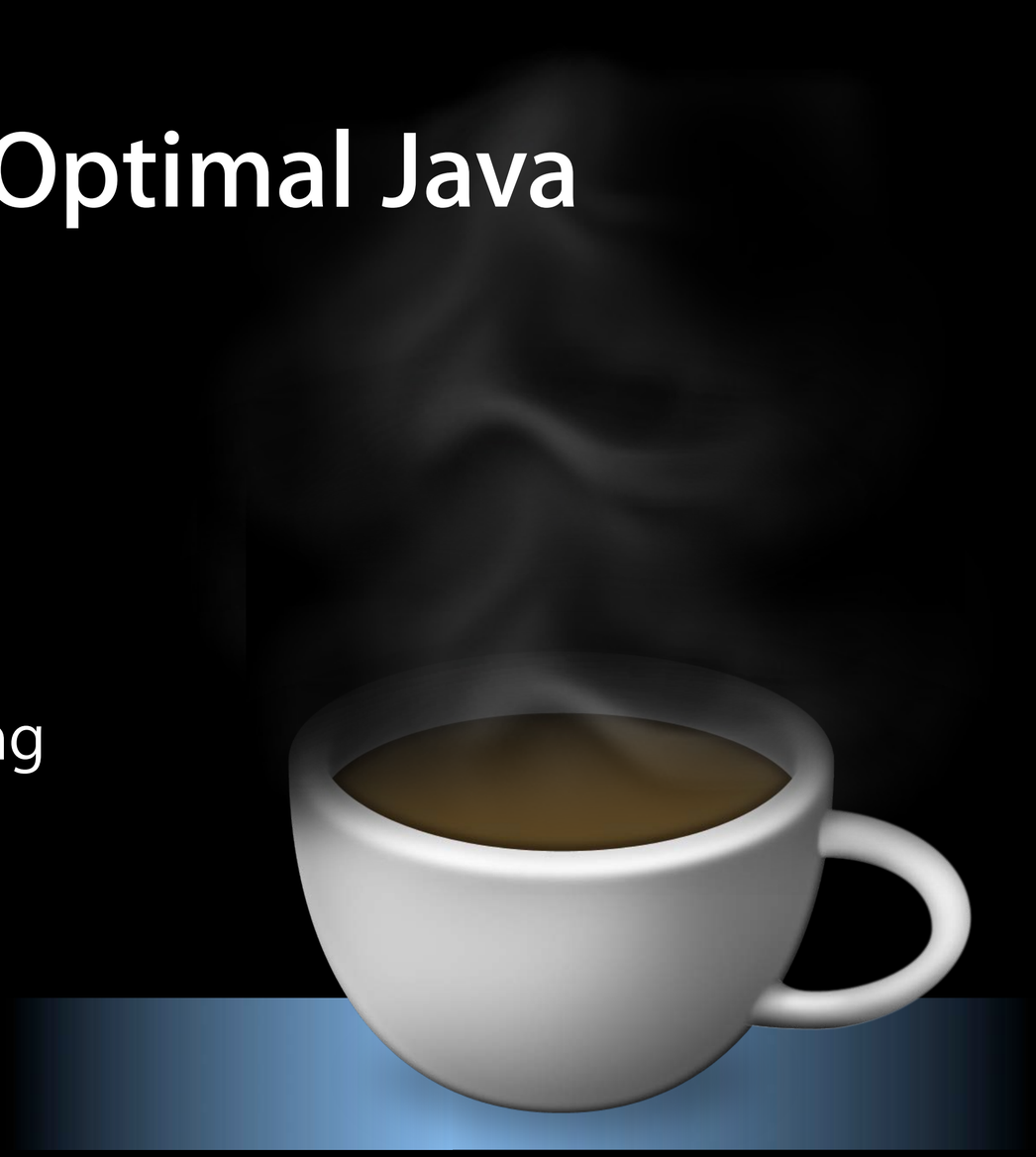
Knowing HotSpot Leads to Optimal Java

Maximize inlining opportunities

- Write small concise methods
- Use accessor methods
- Feel free to use JDK methods
  - Final is not needed for performance tuning

C1 is unable to inline:

- Synchronized
- Large methods
- Methods with exception handlers



Java Tips for the 21st Century

- Exceptions should be exceptional
  - Creating exception objects is expensive
  - Inlining prevented by exception handling
  - Cost only when exception is created
- Avoid object pools
  - Garbage Collection is fast
  - New is inlined
  - Thread-Local allocation reduces contention

Tiger, Java, and You

- J2SE 5.0 on Tiger is ready for your applications
- Tuning for Java
  - Write clean code
  - Let us do the heavy lifting

Java VM on Intel-based Macs

Includes C2 Compiler

Victor Hernandez
Java Virtual Machine Team

http://developer.apple.com/wwdc2005/

Java Virtual Machines in Tiger-based Macs

- Equivalent support for Java 1.4.2 and J2SE 5.0 as in Tiger
- Adding C2 compiler for J2SE 5.0
- High-quality, robust implementation ready for Developer use
- Java applications continue to work
  - All Java dev tools available
  - But applications with native methods will need to be compiled into universal binaries

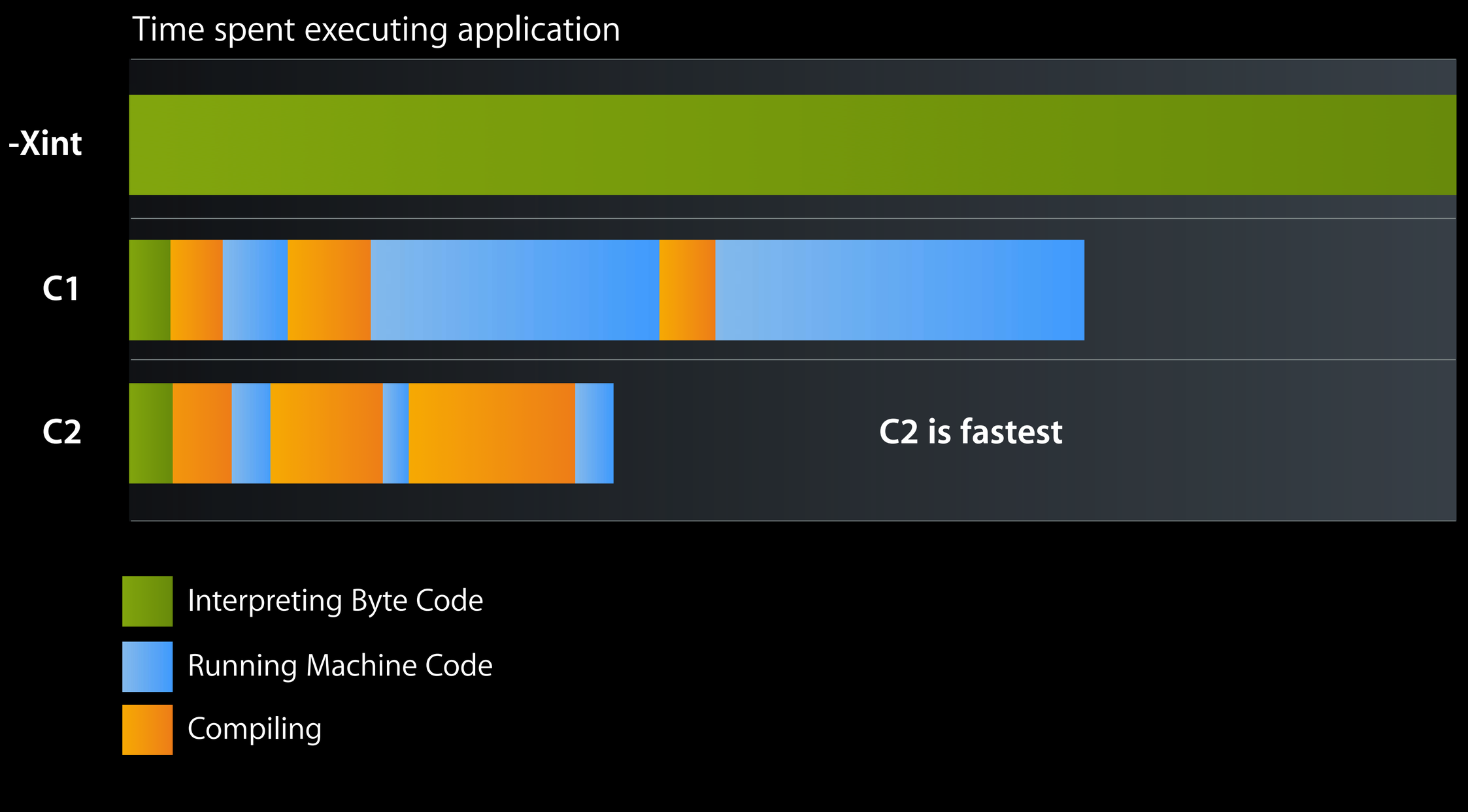
The HotSpot C2 compiler

- Uses more advanced compilation techniques than C1
- Uses the same HotSpot architecture
  - On-stack replacement
  - Dynamic deoptomization
- Is run by adding `-server` to the command line
  - We are working on making it available to bundled applications and applets
- Plugs-in to the JVM just like the HotSpot C1 compiler

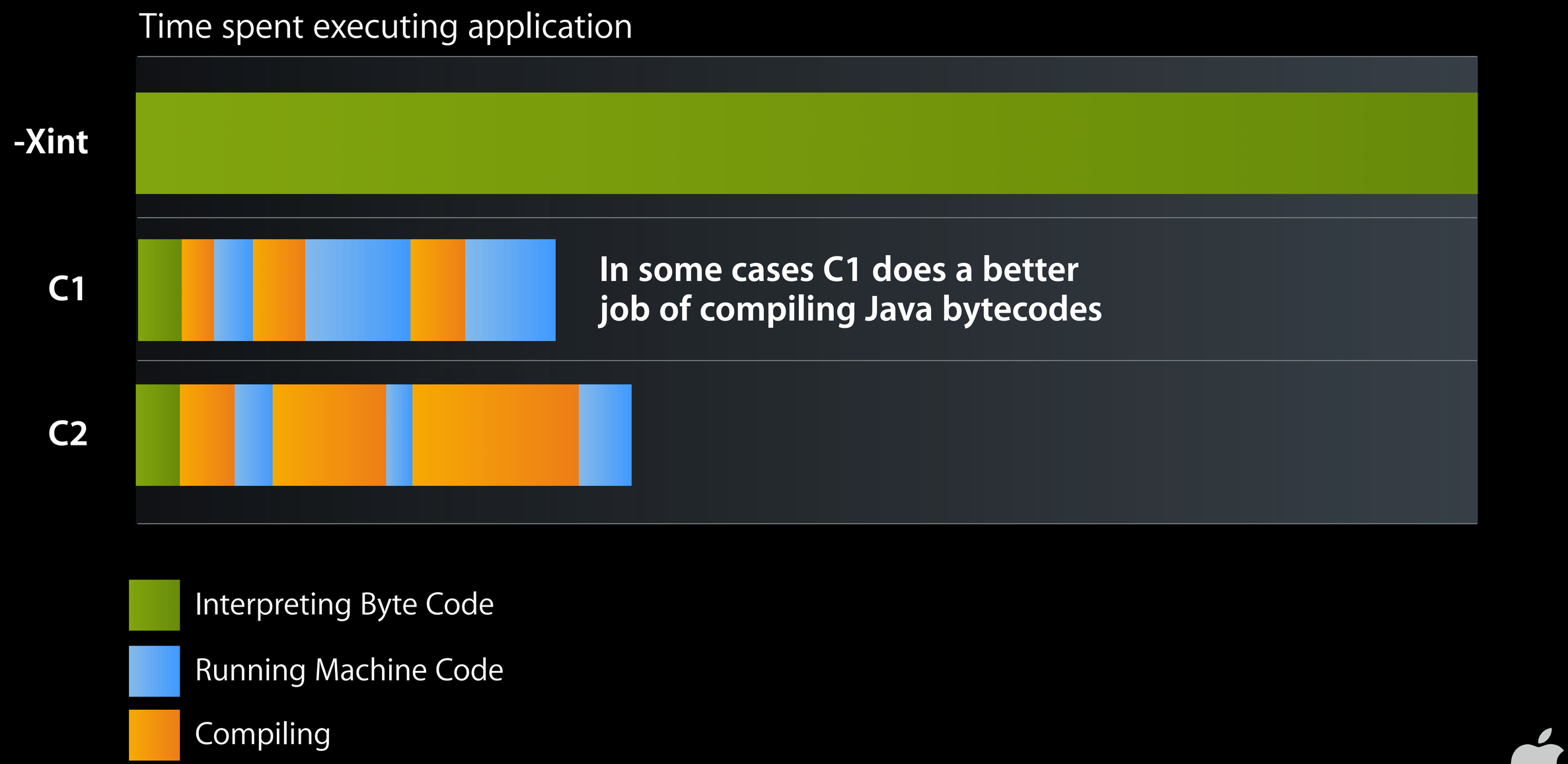
The HotSpot C2 compiler

- Produces better machine code
  - Runs faster
  - But takes a longer time to compile
    - C2 compile time is ~10x more than C1 compile time
  - May cause longer applications pauses
- This is generally a good tradeoff for long-running and compute-intensive programs
- Compilation pauses may be less on multi-processor systems

Pause Times and Run Times



Pause Times and Run Times

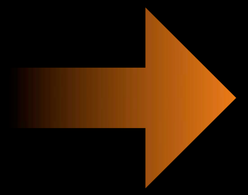


Classical Optimizations Done by C2

- Strength Reduction
- Algorithmic Simplification
- Constant Folding
- Constant Propagation
- Common Subexpression Elimination
- Loop Unrolling
- Loop-invariant Code Motion
- Dead Code Elimination
- Class Hierarchy Analysis
- Aggressive Method Inlining

Strength Reduction

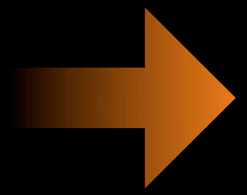
```
a = b / 2
```



```
a = b >> 1;
```

Algorithmic Simplification

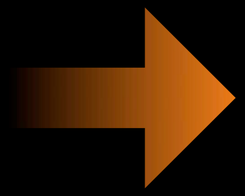
```
a = 1 + (b + 2)
```



```
a = b + (1 + 2)
```

Constant Folding

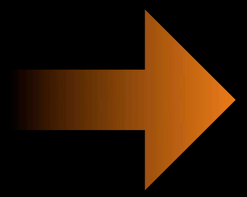
```
a = b + (1 + 2)
```



```
a = b + 3
```

Constant Propagation

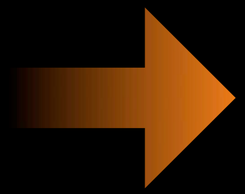
```
a = 5;
b = a + 3;
```



```
a = 5;
b = 8;
```

Common Subexpression Elimination

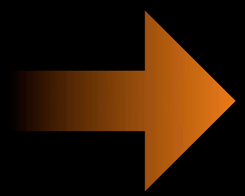
```
x = a + b;
y = a + b;
```



```
x = a + b;
y = x;
```

Loop Unrolling

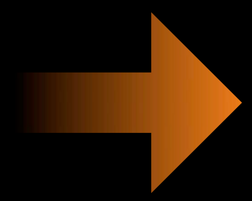
```
for (i = 0 ; i < 4 ; i++)
{
    foo(i);
}
```



```
foo(0);
foo(1);
foo(2);
foo(3);
```

Loop-invariant Code Motion

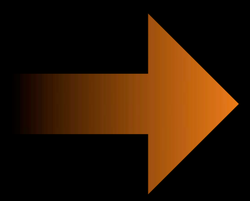
```
for (i = 0 ;
     i < 100 ;
     i++) {
    a[i] = x / y;
}
```



```
int temp = x / y;
for (i = 0 ; i < 100 ; i++)
{
    a[i] = temp;
}
```

Dead Code Elimination

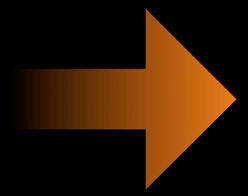
```
if (true) {
  a = 1;
} else {
  a = 2;
}
```



```
a = 1;
```

Class Hierarchy Analysis

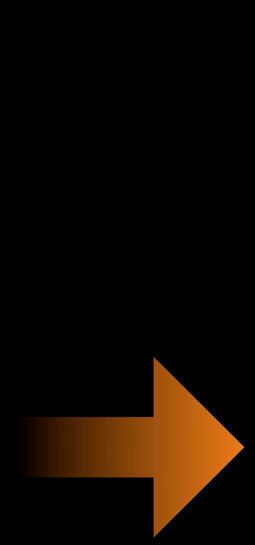
```
int theAnswer()
{
    return 42;
}
```



```
final int theAnswer()
{
    return 42;
}
```

Inlining

```
int average(int a, int b) {
  int s = sum(a, b);
  return s / 2;
}

int sum(int x, int y) {
  return x + y;
}
```



```
int average(int a, int b) {
  int s = a + b;
  return s / 2;
}
```

Inlining in Hotspot often better than in static compilers

Putting it All Together

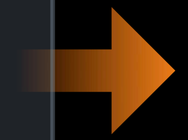
```
static int foo(int n, int m) {
    int x;
    switch(n) {
        case 1:  x = 1*m; break;
        case 2:  x = 2-m; break;
        case 3:  x = 3+m; break;
        default: x = 4/m; break;
    }
    int y = x;
    int z = y * 123;
    if (y > 50) {
        return ((y+m)<<n)-358;
    } else {
        return z;
    }
}

...

System.out.println(foo(1, 100));
```

Putting it All Together

```
static int foo(int n, int m) {      // n = 1, m = 100
    int x;
    switch(n) {
        case 1:  x = 1*m; break;    // x = 100
        case 2:  x = 2-m; break;
        case 3:  x = 3+m; break;
        default: x = 4/m; break;
    }
    int y = x;                      // y = 100
    int z = y * 123;
    if (y > 50) {                   // if (true)
        return ((y+m)<<n)-358;      //   return ((100+100)<<1)-358
    } else {
        return z;
    }
}

...

System.out.println(foo(1, 100));
```

Putting it All Together

```
static int foo(int n, int m) {      // n = 1, m = 100
    int x;
    switch(n) {
        case 1:  x = 1*m; break;    // x = 100
        case 2:  x = 2-m; break;
        case 3:  x = 3+m; break;
        default: x = 4/m; break;
    }
    int y = x;                      // y = 100
    int z = y * 123;
    if (y > 50) {                   // if (true)
        return ((y+m)<<n)-358;      //   return ((100+100)<<1)-358
    } else {
        return z;
    }
}

...

System.out.println(foo(1, 100));
```



```
System.out.println(42);
```

Other C2 Features

- Graph coloring register allocator
- Uses SSA IR
  - Static Single Assignment Intermediate Representation
  - Makes previous operations easier
- Null check elimination (mainly useful for other optimizations)
- Best instruction selection for processor
- More aggressive use of Java memory model

Avoid Premature Optimizations

- C2 is really good at many optimizations
  - Let it do so
  - Keep your code clean

- Exceptions should be exceptional
- No need to use `final` to speed up your code
- Object pools are only beneficial for very large objects

Java Performance

C1 Compiler vs. C2 Compiler

Roger Hoover
Java Virtual Machine Team

http://developer.apple.com/wwdc2005/

Introduction

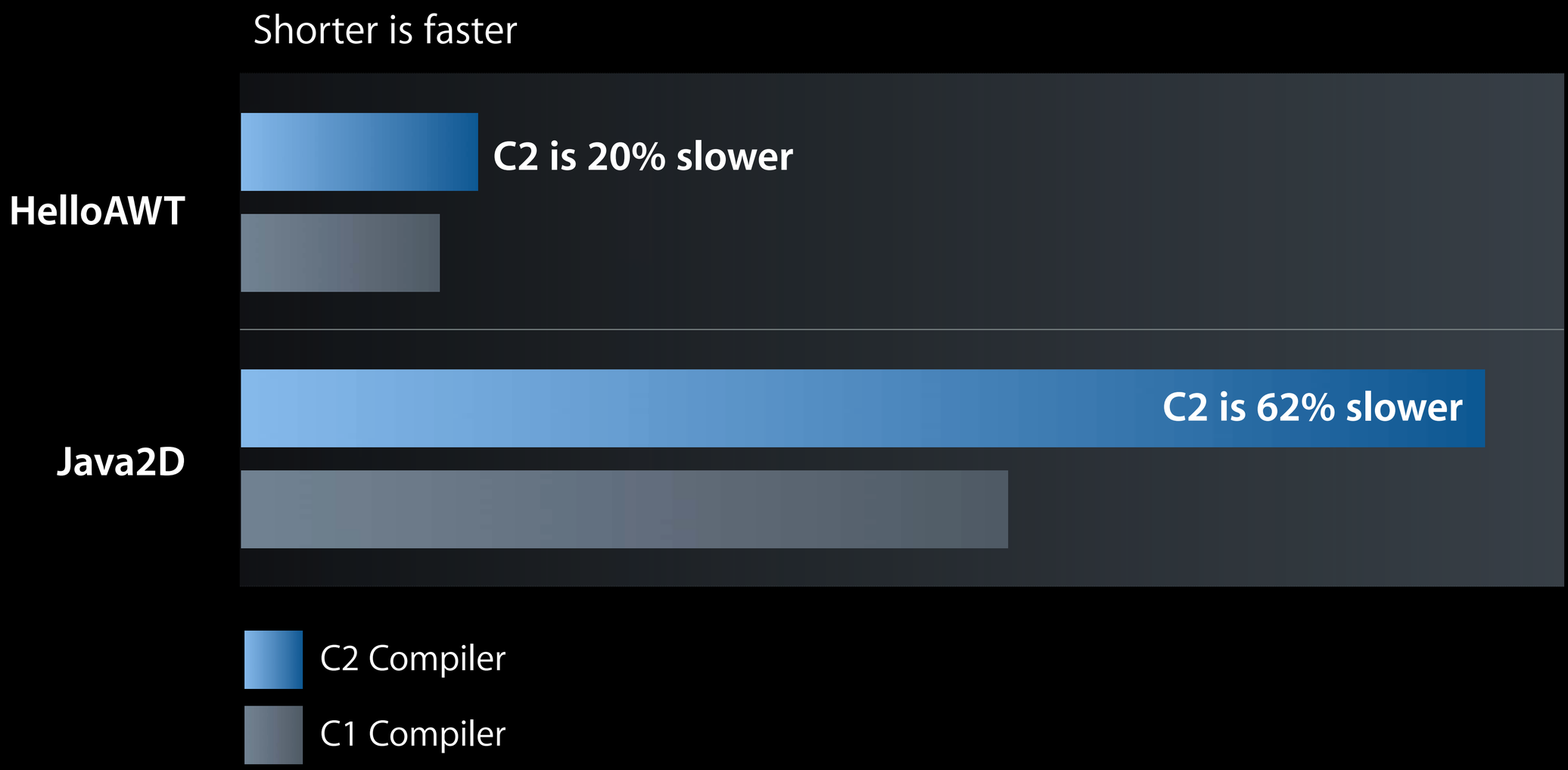
- How does performance change with new JVM?
  - C1 vs. C2 Compilers
  - Intel-based Macs
- Measurements based on
  - Startup time of GUI application
  - Footprint of GUI application
  - JavaVM Benchmarks

Preliminary Performance Results

- No time spent tuning Intel Java VM
- Known performance issues that we will fix

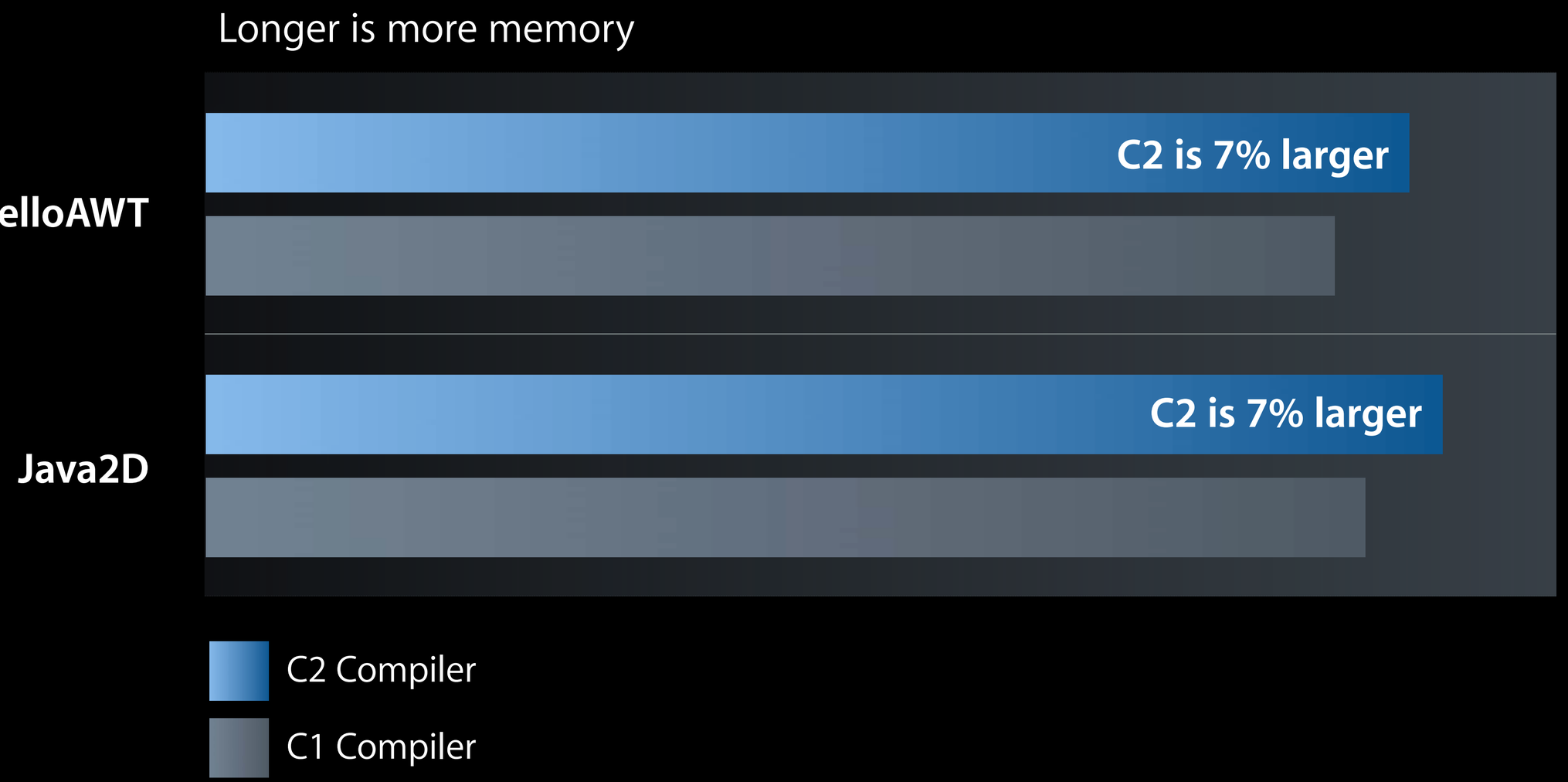
C1 Compiler vs. C2 Compiler

Startup Time



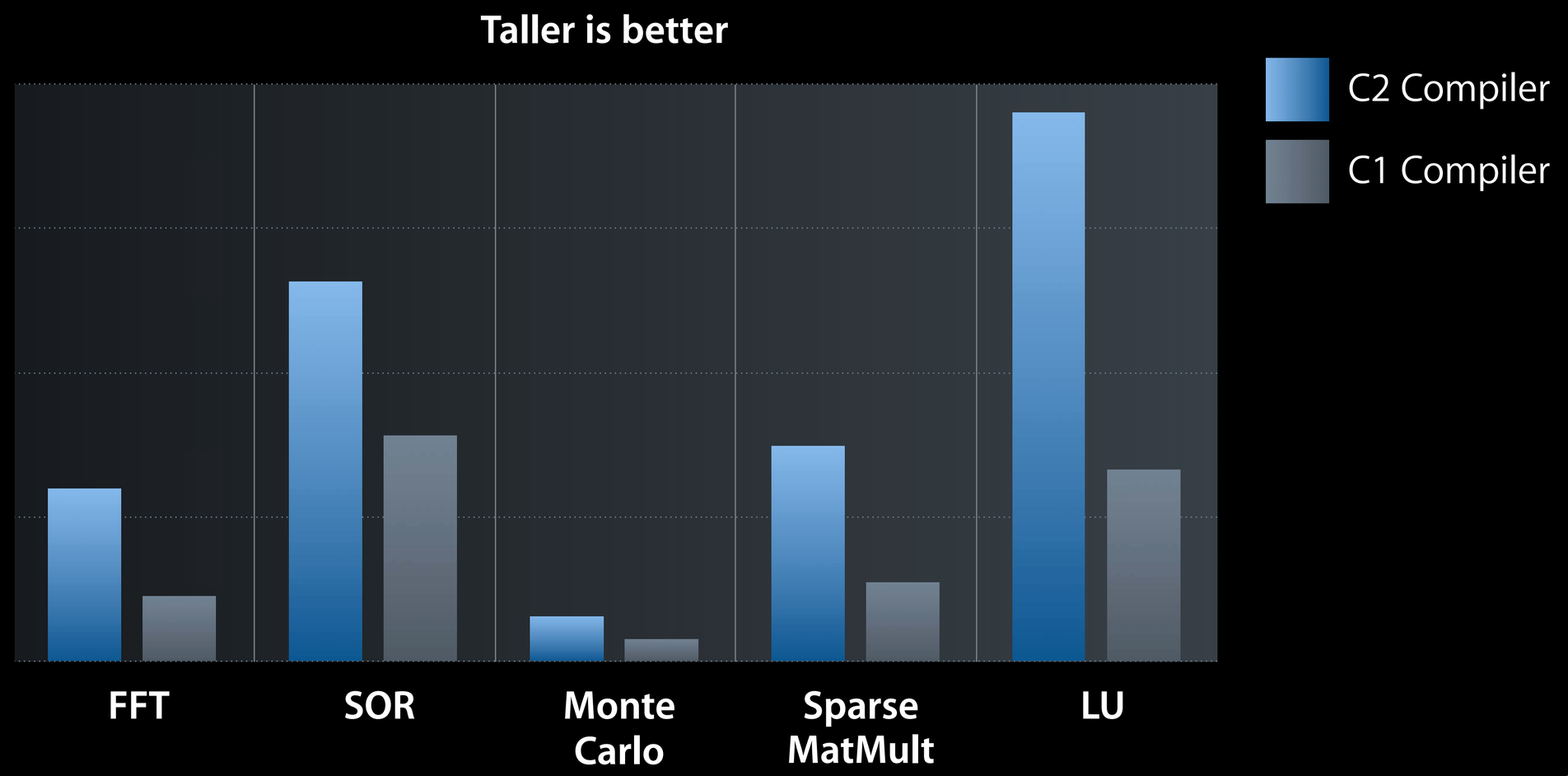
C1 Compiler vs. C2 Compiler

Footprint Cost



Scimark 2.0

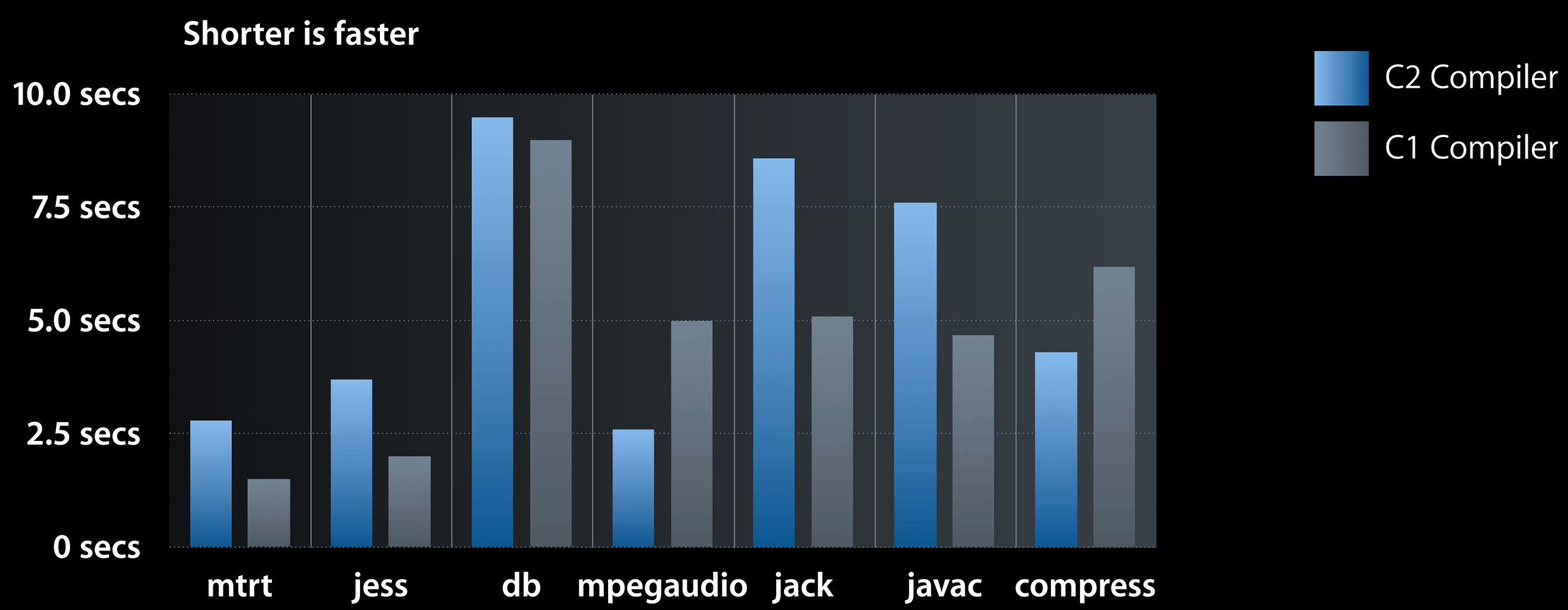
Scientific Math Benchmark



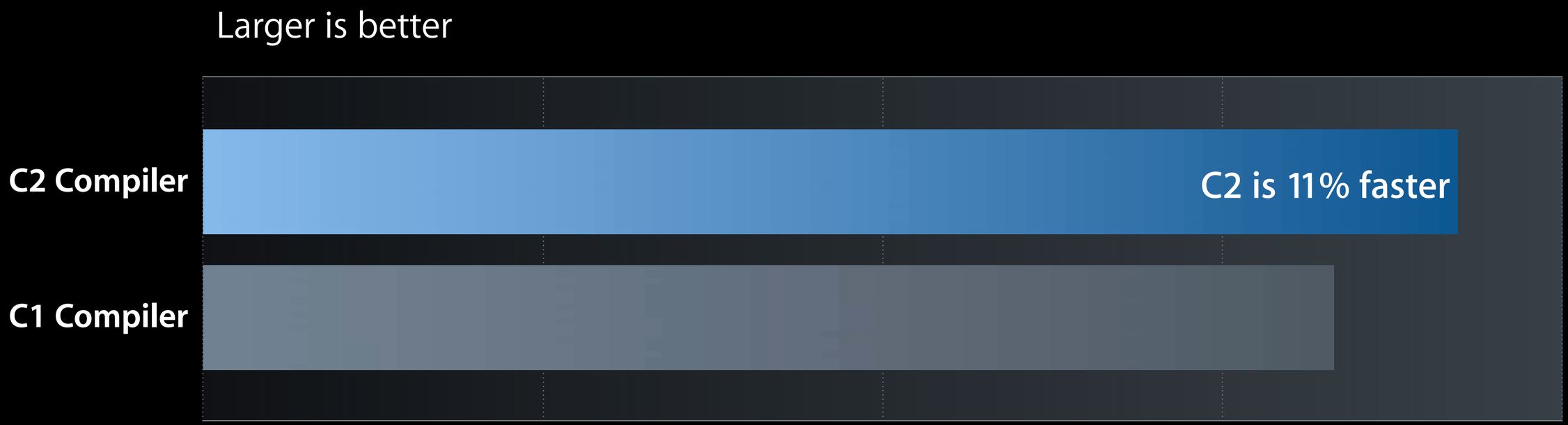
C2 is 2.3x faster!

Well-known Benchmarks

C2 Compiler does not always beat C1 Compiler

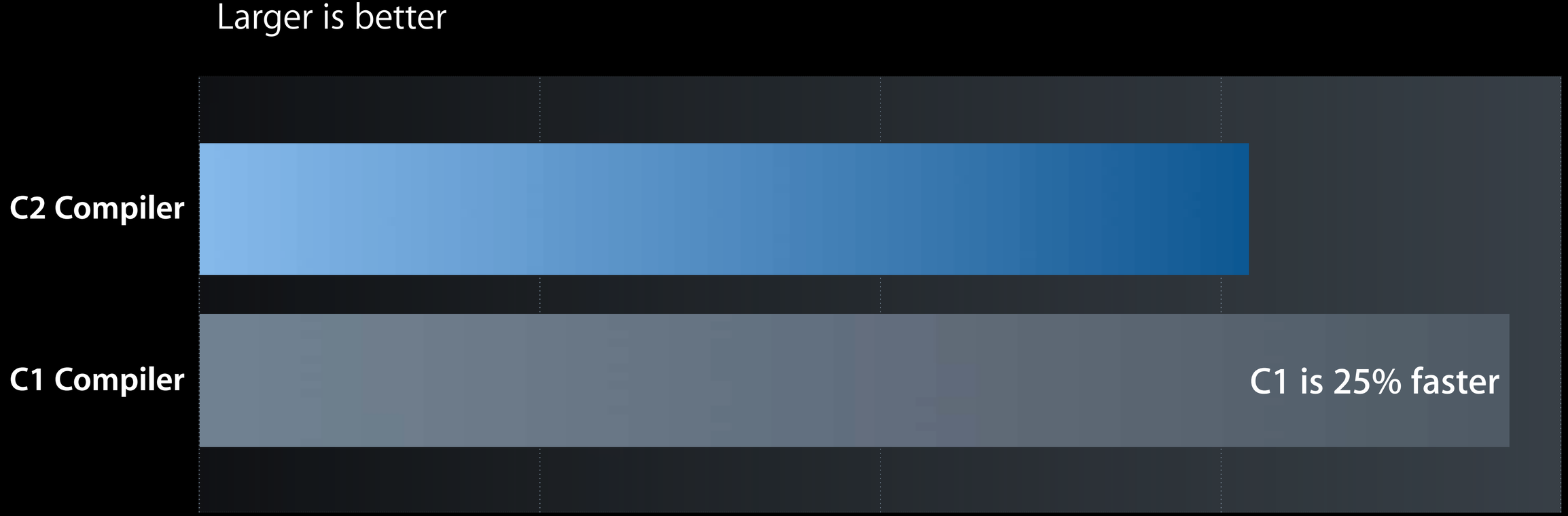


Benchmark of Business Logic



VolanoMark

I/O-bound Client/Server Benchmark



Performance Conclusions

- These are exciting times!
- Faster performance with C2 compiler
- But C2 does not always win

We Want your Feedback

Java and WebObjects Feedback Forum	Session 641 Friday 3:30PM

Who to Contact

Alan Samuel
Java Evangelist
bluker1@apple.com

François Jouaux
Java Technologies Manager
fjouaux@apple.com

Q&A

Alan Samuel
Java Evangelist
bluker1@apple.com

http://developer.apple.com/wwdc2005/

WWDC2005

Worldwide Developers Conference